Proof of Japanese Multiplication in ACL2s

Karim Semaan
Khoury College of Computer Sciences
Northeastern University
Boston, MA, United States
semaan.k@northeastern.edu

Elisha Mann-Robison
Khoury College of Computer Sciences
Northeastern University
Boston, MA, United States
mann-robison.e@northeastern.edu

Dominik Ritzenhoff
College of Engineering
Northeastern University
Boston, MA, United States
ritzenhoff.d@northeastern.edu

Ben-oni Vainqueur
College of Engineering
Northeastern University
Boston, MA, United States
vainqueur.b@northeastern.edu

Jason Hemann
Khoury College of Computer Sciences
Northeastern University
Boston, MA, United States
jhemann@northeastern.edu

Table of Contents

What is Japanese Multiplication? 3

Code Walkthrough 5

Normal Multiplication 5

Program Structure: 5

Japanese Multiplication 6

Data definition: 6

Functions: 6

Program Structure: 6

The Undiscovered Lemma 7

Conclusion 11

Appendix 12

Code for Japanese Multiplication 12

Code with Defthms 12

What is Japanese Multiplication?

Japanese Multiplication is an ancient method of finding the product of two numbers using lines and intersections. Although not commonly taught, Japanese Multiplication is significant because it is one of the few methods that doesn't need the 0-9 number system to function. Within the following paragraphs, we will describe how this method works and how we represented this algorithm within ACL2 to prove its equivalence to the 'normal' multiplication operator.

In order to perform computations using Japanese Multiplication, we draw a series of lines corresponding to each digit and count the total number of intersections. Let's take 34 x 12 as an example. Beginning with 34, we draw a set of 3 and 4 adjacent parallel lines. Seen in Fig. 1, the first three lines each represent a value of 10 while the 4 other lines each represent a value of 1. Notice how this is the same as decimal notation. Fundamentally, what we are doing is simply translating from one form (numbers) to the next (lines). For the number 12, we draw one line, leave a small gap, and then draw two parallel adjacent lines. The final result is an intersection of lines to form a grid-like pattern. This can be seen in Fig. 2.

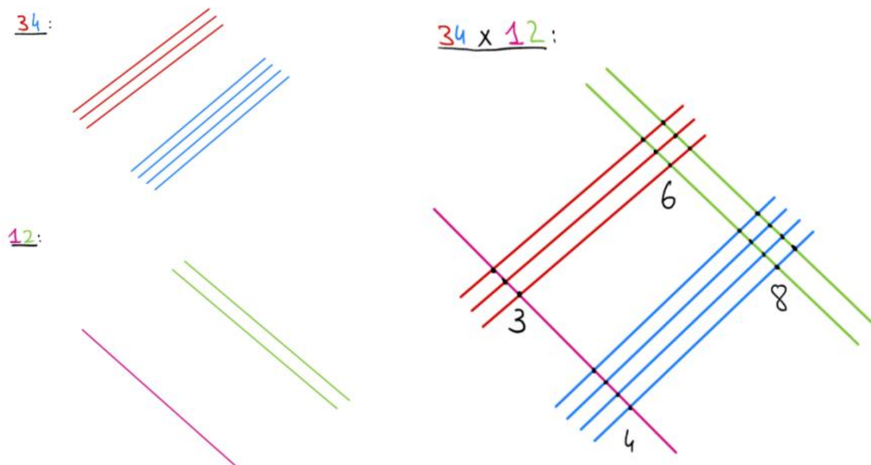


Fig. 1 – Line representation of numbers **Fig. 2** – Overlapping the parallel lines

At this point, we begin counting the total number of intersections and group the results. We do so by drawing a loop around every set of intersections closest to the left side. Within Fig. 2, this is where the red and magenta lines intersect. Then, we proceed to the right until we process all the intersections. As a caveat, should two or more sets of 'dots' be vertically aligned, they exist within the same loop. Seen in Fig. 3, this method produces a total of three groups, each of which represents the hundred's, ten's or one's place. Finally, we get the product by taking the sum of these values:

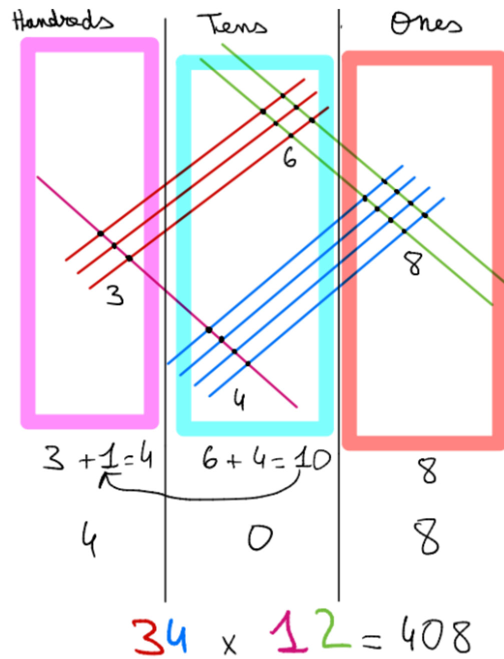


Fig. 3 – Final Groupings of Numbers

Like in normal addition, we 'carry the one' should the number be 10 or greater. As a result, this produces 4 hundreds, 0 tens and 8 ones. After adding these values together, we get 408, which is the product. This is how Japanese multiplication functions.

Now, with respect actually modeling this method within ACL2, we chose to treat 34 and 12 as lists of digits. More specifically, 34 and 12 would be represented as [4, 3] and [2, 1], respectively. The reason why each list seems backwards is to make the multiplication process easier on ourselves, which will become clear later in the paper. For now, though, there are a few implications to note because of this design choice:

1) It would be difficult to limit our `defdata` (`list-of-nats` and `list-of-list-of-nats`) to strictly 1-digit naturals ([1, 2, 3, 4] vs. [12, 34]), so we chose to let the data definitions be whichever natural numbers they are assigned, regardless of size. The correct result will still be produced.
2) Converting from list form into number form is done as such:

$$\sum_{i=0}^{k} d_i \rightarrow 10^0 * d_0 + 10^1 * d_1 + \cdots + 10^i * d_i$$

This method of conversion actually has a few implications itself worth mentioning. First, because the digits are not always numbers from $0 - 9$, it is unnecessary to limit our base to 10. Furthermore, why would we? Using the conversion equation we just presented, simply multiplying our digits by some given base b would convert the number into its proper value under that given base. Our `list-to-num` conversion changes to take the following form:

$$\sum_{i=0}^{k} d_i \rightarrow b^0 * d_0 + b^1 * d_1 + \cdots + b^i * d_i$$

However, it must also be noted that b^0 is the coefficient for d_0, implying that d_0 represents the first and smallest digit within our number. In order for this to be the case, our given numbers must not only be lists of digits, but they must be **in reverse order**. For example, $321_{10} \rightarrow [1,2,3]$ to make our conversion work.

Code Walkthrough

The codebase can be thought of as two separate components. The first deals with 'normal' multiplication (i.e., $4 * 5$) while the second concerns itself with Japanese multiplication. Beginning with the former component, the regular multiplication uses the `normal-mult` function and takes two reversed list representations of a number (i.e., $123 \rightarrow [3,2,1]$) and a base as inputs. The lists are translated back into their natural forms with a helper function, and the product of the resulting two numbers is returned.

Normal Multiplication

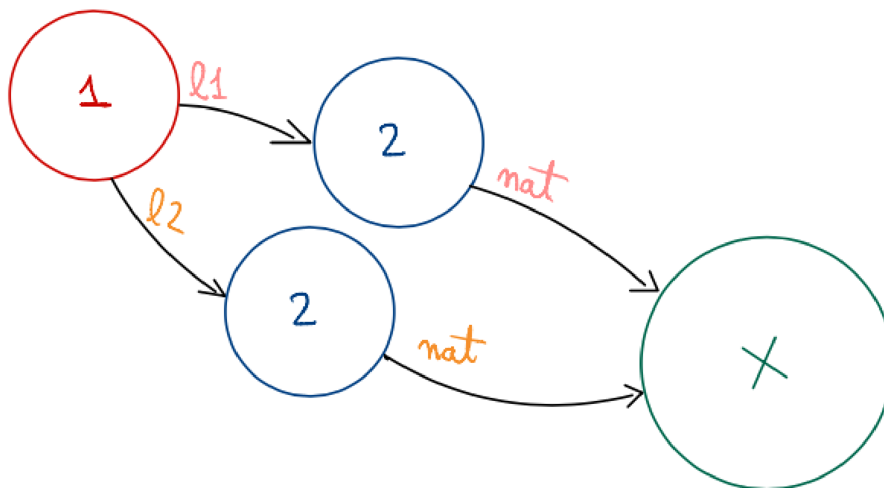
Data definition:

- `list-of-nat` → `lon`
- `list-of-list-of-nat` → `lolon`

Functions:

1) `normal-mult (l1: lon, l2: lon, b: nat)` → `nat`
2) `list-to-num: (lst: lon, b: nat)` → `nat`

Program Structure:



`List-to-num`, or function 2 in the figure above, multiplies a list such as $[3,2,1]$ by sequential powers of the base starting with 0. Therefore, assuming a base of 10, the function would return the sum of $3 * 10^0 + 2 * 10^1 + 1 * 10^2$, which is 123.

Unlike normal multiplication, Japanese multiplication makes more careful use of the form that the two numbers are given in. Since we are given the digits in reverse order, we can treat these numbers as operatable vectors. With this interpretation of our data's formatting, we can create a matrix out of component-wise multiplication between the vectors. Suppose we are given numbers c and d represented as follows:

$$c = [c_0, c_1, \dots, c_i], d = [d_0, d_1, \dots, d_j]$$

The matrix we can construct would look like this:

$$c \times d = \begin{bmatrix} c_0 * d_0 & c_0 * d_1 & \dots & c_0 * d_j \\ c_1 * d_0 & c_1 * d_1 & \dots & c_1 * d_j \\ \vdots & \vdots & \vdots & \vdots \\ c_i * d_0 & c_i * d_1 & \dots & c_i * d_j \end{bmatrix}$$

If we compare this matrix to Fig. 2's visual multiplication (i.e., using $c = 34, d = 12$), we begin to see how the above matrix represents the same operations as shown in Fig. 2. Every index of our matrix – $c_k * d_g$ – matches up to an intersection between two digits in Fig. 2, and the resulting value is the same. In fact, we could represent Fig. 2 as such:

$$[4,3] \times [2,1] = \begin{bmatrix} 4*2 & 4*1 \\ 6*2 & 3*1 \end{bmatrix} = \begin{bmatrix} 8 & 4 \\ 6 & 3 \end{bmatrix}$$

The above matrix is a rotation of Fig. 2 (since the numbers in Fig. 2 have their digits in non-reversed order). However, the next step complicates our matrix, because we don't grab the rows or the columns but rather the diagonals. As seen in Fig. 3, the diagonals are grouped together and then summed. The simplest method that we found to simulate this in our code was to "push" rows so that the diagonals turned into the columns. For example, if we are using the matrix from above, ($\begin{bmatrix} 8 & 4 \\ 6 & 3 \end{bmatrix}$), then the ultimate list of lists that we want is this: $[\,(8),(4,6),(3)\,]$. To make these lists, we separate each grouping into its own column by padding the beginning and endings of certain rows with 0's:

$$\begin{bmatrix} 8 & 4 \\ 6 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 8 & 4 & 0 \\ 0 & 6 & 3 \end{bmatrix}$$

As you can see, the columns now contain what originally were the diagonals, and allow us to easily construct the list of lists that we wanted. Then, the next step involves 'flattening' this list-of-list-of-nats by summing the elements in each list to get their sums. Lastly, this list of numbers is transformed back into a number and is returned as our final result. For example:

$$\mathbf{12 \times 34}: [\,(8),(6,4),(3)\,] \rightarrow [\,8,10,3\,] \rightarrow 300 + 100 + 8 = 408$$

Japanese Multiplication

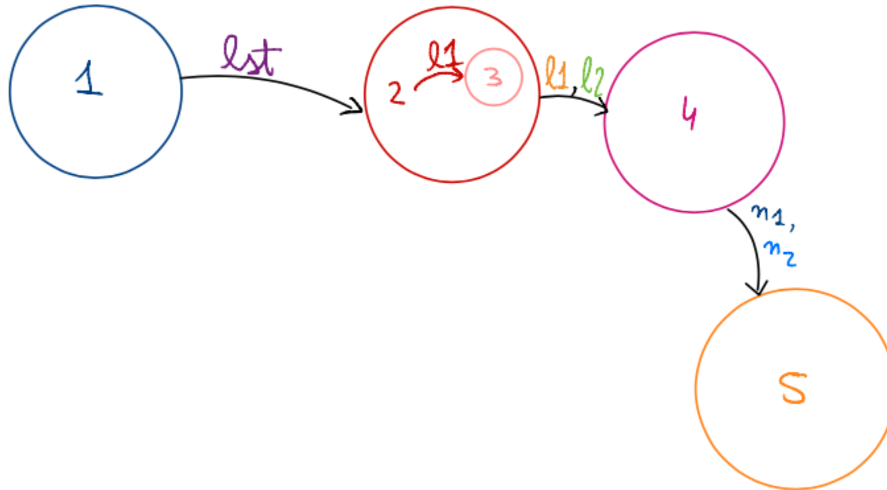
Data definition:

- `list-of-nat →  lon`
- `list-of-list-of-nat →lolon`

Functions:

1) `list-to-num (lst: lon, b: nat) →nat`
2) `flattenlolon (l1: lolon) →lon`
3) `flattenlon (l: lon) →nat`
4) `diagonalize (l1: lon, l2: lon) →lolon`
5) `jmult (n1: lon, n2: lon, b: nat) →nat`

Program Structure:



With respect to the actual functions, Japanese multiplication begins with '`diagonalize`', which takes that list representation of numbers and translates it into a padded list of lists of naturals similar to the matrix seen below. Remember, the objective of this function is to group the elements that are diagonal from one another together.

$$\begin{bmatrix} 8 & 4 \\ 6 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 8 & 4 & 0 \\ 0 & 6 & 3 \end{bmatrix}$$

Next, the list is consolidated with a call to the '`flattenlolon`' function. This is equivalent to the matrix being compressed down into a 1-dimensional form, which is accomplished by adding the paired values together. So, in the example above, $8 + 0 = 8$, $4 + 6 = 10$, and $0 + 3 = 3$. This produces the following list.

$$[8 \quad 10 \quad 3]$$

Finally, and much like the '`normal-mult`' function, '`list-to-num`' is called to translate the list representation of a number into its natural form. This is done by multiplying the list by sequential powers of the base starting with 0: $8 * 10^0 + 10 * 10^1 + 3 * 10^2 = 408$.

The Undiscovered Lemma

Now, after an introduction to the inner workings of both multiplication methods, we will move on to prove the equivalence between them. This, after all, is what the essay is entirely about. In the following paragraphs, we walk through the avenues we took to prove this lemma and the many struggles we faced while doing it.

To start, we wrote tests for every function to verify they were behaving as expected. After all the tests passed, the next step was to use ACL2's theorem-proving ability to prove beyond a doubt that the two processes were equivalent. For this, we constructed the following theorem:

```
(defthm jmult-equiv
  (implies (and (lonp x) (lonp y) (natp z))
           (equal (normal-mult x y z) (jmult x y z))))
```

When we ran this through ACL2, we immediately encountered a problem: ACL2 needed more information about the nature of our code before it could prove this equivalence. In essence, we needed more theorems (although really, they would all just be lemmas)!

However, we made the realization that throwing random theorems at the wall wouldn't necessarily help ACL2 move forward with its proof, and it was also not sustainable. This therefore pushed us towards a systematic approach of determining lemmas to help prove our ultimate theorem. After hours of work and the development of an extreme love/hate relationship with the Proof Checker, we gained a bit of intuition: substitute. In nearly every proof that we had done in the proof checker, the steps amounted to a series of substitutions. So, we began to substitute into our theorem's statement to see if we could determine where ACL2 was getting hung up.

It wasn't long before we found the bottleneck (luckily for us, using `(set-gag-mode nil)` helped us find it in the output of ACL2 itself): ACL2 could get to the following lemma, but couldn't prove it:

```
(IMPLIES (AND (AND (LONP X) (LONP Y))
              (NOT (ENDP Y))
              (EQUAL (* (LIST-TO-NUM X Z)
                        (LIST-TO-NUM (CDR Y) Z))
                     (LIST-TO-NUM (FLATTENLOLON (DIAGONALIZE X (CDR Y)))
                                  Z))
              (LONP X)
              (LONP Y)
              (NATP Z))
         (EQUAL (* (LIST-TO-NUM X Z) (LIST-TO-NUM Y Z))
                (LIST-TO-NUM (FLATTENLOLON (DIAGONALIZE X Y))
                             Z)))
```

There was a lot of information here, but the important nature of this statement was clear: if we could prove this to ACL2, we would be one step closer to proving the equivalence between Japanese multiplication and standard multiplication. Thus, we began by working to expand the first half of the following statement:

```
(* (list-to-num x z) (list-to-num y z))
```

Using the following two lemmas, we managed to replace it with a more informative statement, which would hopefully let the proof checker pass:

```
(defthm scale-axiom
   (implies (and (lonp x) (lonp y) (natp z) (not (endp y)))
            (equal (* (list-to-num x z) (car y))
                   (list-to-num (scale x (car y)) z))))

(defthm distr-list-to-num
  (implies (and (lonp x)
                (lonp y)
                (natp z))
           (equal (* (list-to-num x z) (list-to-num y z))
                  (+ (* (list-to-num x z) (car y))
                     (* z (* (list-to-num x z) (list-to-num (cdr y) z)))))))
```

As you can see, our second lemma, `distr-list-to-num`, gives us a direct substitution for the first half of our original statement. We thought of this as a sort of middle ground between the two initial statements we had in our subgoal. If we could get the second half of our subgoal to look like this adaptation of the first then by transitivity, we would have proven our subgoal!

So, we set to work attempting to wrangle this statement into shape:

```
(list-to-num (flattenlolon (diagonalize x y)) z)
```

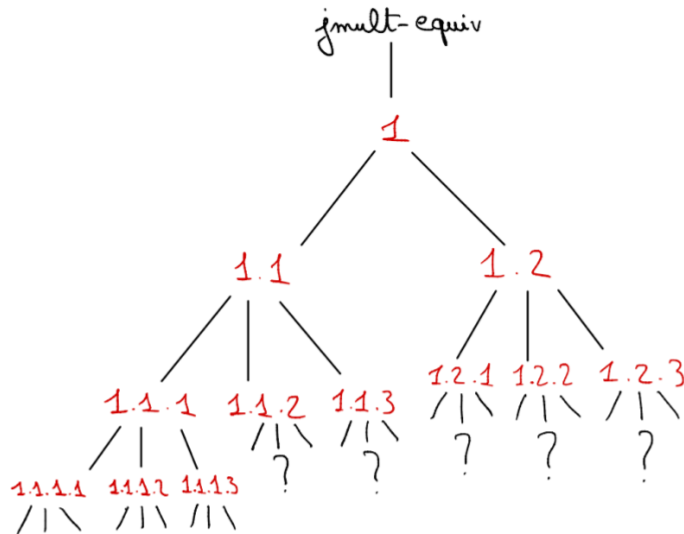
As before, our first step was to substitute. This time, we substituted the function call to diagonalize for its function body. However, we quickly ran into an issue: this subgoal assumed that y was not empty, but we needed more information on x in order to reason about it. To this end, we created an additional lemma:

```
(defthm almost-jmult-equiv
  (implies (and (lonp x) (lonp y) (natp z) (endp x))
           (equal (normal-mult x y z) (jmult x y z))))
```

After confirming that ACL2 could prove this, we now had gained information on x: if x were empty, the above theorem told us that it had to be true. Therefore, the only unproven values of x that might not work would be if it were <u>not</u> empty. By that reasoning, we could modify ACL2's subgoal to include the context that x was not empty.

However, when we tried to prove this `defthm` for the cases in which `jmult` was not empty, we realized that it was leading to dead ends in which there were subgoals expanding into multiple other subgoals. This made for an arduous proving process and ultimately could never finish. Because of this, we decided to use a different technique altogether. This time, we attempted using skip-proofs. This was accomplished by running the `jmult-equiv` function until ACL2 failed. After that, we looked at all the subgoals that were output and skipped their proofs to check if `jmult-equiv` could pass. Surprisingly, it did! This simplified the problem greatly; now, all we had to do was prove those subgoals, and we could prove the lemma. This approach fundamentally made more sense because this time, we were giving ACL2 what it wanted instead of trying to find a `defthm` that could miraculously prove our lemma.

However, we ran into a similar problem as before because the two subgoals meant to prove the lemma kept expanding into multiple other subgoals. This image illustrates the problem:



After making this realization, we started from scratch yet again and reran the `jmult-equiv` theorem but added hints along the way that dealt with what to induct on. For example, we added the following: `:hints (("Goal" :induct ((tlp x)))` and `:hints (("Goal" :induct ((diagonalize x y)))` and `:hints (("Goal" :induct ((Scale Y Z)))`. Although promising, this led us into the same expanding-subgoal problem as before, which forced us to drop the approach.

For our fourth attempt, we thought of proving the `jmult-equiv` theorem by using a 'bridge function'. Inspired by the transitive property, we thought that if we could prove '`jmult`' is equal to 'bridge' and 'bridge' is equal to '`normal-mult`', '`jmult`' must be equal to '`normal-mult`'. Within this context, 'bridge' would represent distributive multiplication, which is yet another method to calculate the product of two numbers. As a result, we added `distr-mult-equiv` and `distr-jmult-equiv` to prove equivalency between the 'bridge' and the two other multiplication methods.

Proving to ACL2 that distributive multiplication was equivalent to Japanese multiplication was easier than we thought, but it forced us to use every already-mentioned strategy. Our first step involved running the proof through ACL2 and testing to see where it got stuck. ACL2, after some time, returned two subgoals that it was getting fixed on. We then employed our skip proofing method to confirm that the two given subgoals, if proven, would prove the ultimate theorem. After seeing this was the case, we examined the subgoals we were given, and we simplified and rewrote them until we could understand where ACL2 was getting hung up. This revealed that ACL2 was getting stuck on the interactions between two of our functions: `flattenlolon` and `append-lon`. We knew exactly how to reason about those functions, but we had not yet explained to ACL2 how to think about them. In order to solve this problem, we wrote a simple lemma relating (and simplifying) the two, so that ACL2 could pull apart and reason about expressions involving these functions. This was the final roadblock, and we were finally able to prove the theorems. At last, the undiscovered lemma had been found.

Conclusion

The concrete takeaway from our proof is that there is more than one way to represent the process of multiplication. Be it with standard multiplication [1], which is the societal norm, or with 'sticks', which Japanese Multiplication utilizes, we got to the same product both ways. This is important to note because it demonstrates the abstract nature of mathematics. The notation that we humans choose to accept isn't necessarily 'correct'; it's simply one of the many ways we can visually model quantities and quantity manipulations.

With that, it's important to further dive into the differences between multiplication methods and their limitations. With respect to computers, more specifically, arithmetic operations must be done as quickly as possible, which necessitates a multiplication method that is maximally efficient, both spatially and timewise. Importantly, it must be noted that Japanese multiplication is none of those things when compared to other algorithms like Karatsuba's [2]. Our proof therefore doesn't necessarily improve on the process of multiplication with respect to time and space, but it eliminates Japanese multiplication's use case within this realm of computation. Put simply, the algorithm is too slow and is too memory inefficient. This, in turn, strengthens the credibility of the most efficient algorithms today.

Nevertheless, our group succeeded because we were able to prove equivalency between the `j-mult` and `normal-mult` functions. Moreover, our results proved equivalence between all numbers across all bases ($n > 0$), generalizing the use of Japanese Multiplication such that it can be used with a wide range of bases, rather than just base 10.

Appendix

[1]: “Multiplication algorithm,” *Wikipedia*, 04-Apr-2021. [Online]. Available: https://en.wikipedia.org/wiki/Multiplication_algorithm. [Accessed: 26-Apr-2021].

[2]: “Karatsuba algorithm,” *Wikipedia*, 07-Apr-2021. [Online]. Available: https://en.wikipedia.org/wiki/Karatsuba_algorithm. [Accessed: 26-Apr-2021].

Code for Japanese Multiplication

https://github.com/benonivainqueur/Japanese-Multiplication/blob/main/jmult-no-defthm.lisp

Code with Defthms

https://github.com/benonivainqueur/Japanese-Multiplication/blob/main/jmult.lisp